White Paper

Intel Corporation

Data Prefetching

Optimizing Embedded System Performance—Impact of Data Prefetching on a Medical Imaging Application

Embedded system developers may find added performance for particular workloads when hardware and software data prefetching is finely tuned.

TABLE OF CONTENTS

Executive Summary

Application software optimization is seldom a smooth and straight forward process. Software developers must often experiment with a range of techniques and configurations in order to wring performance out of their system. Fueling the complexity is advanced CPUs with multiple cores, large caches, and hardware prefetchers, making the software optimization process anything but simple and intuitive.

Perhaps one of the least understood aspects of modern CPU architecture is the interaction between the cache memory and hardware prefetch mechanisms. High-performance CPUs rely heavily on cache memory to hide memory latency. To make caches more efficient, hardware prefetch units preemptively load data into caches in anticipation of using this data to process subsequent instructions.

While most workloads experience higher performance because of the hardware prefetcher, some workloads can cause the prefetcher to guess incorrectly and populate the processor's caches with unneeded data. For some workloads and data structures, hardware prefetching may actually lower performance. This paper examines the impact of hardware and software data prefetching on the AMIDE* medical imaging application running on Intel dual-core processors. When this application ran with hardware prefetching enabled, it was three times slower than when software prefetch instructions were implemented.

Performance Improvements Derived from Cache Memory

The performance of most CPUs is highly dependent upon the availability of data and instructions to the execution unit. To lower data latency, the execution unit is surrounded by high-speed static RAM (SRAM) known as cache memory. Storing frequently used data close to the CPU minimizes the need for the CPU to fetch data from system memory, avoiding significant delays.

A block diagram of the Intel Core 2 Duo processor and its cache implementation is shown in Figure 1. This represents a dual-core processor with one core on the right side and another core on the left side. The two cores share the L2 cache shown in the center.

Each core has three dedicated hardware prefetchers: one is located in the Instruction Fetch and PreDecode block (#1, #4) and the other two are in the pipelines between the Schedulers and the L1 Data Cache (#2, #3, #5, #6). Two additional prefetchers are dynamically shared between the two cores, and these are located in the L2 cache (#7, #8).

The cache and prefetching implementation shown in Figure 1 can improve data latency as much as eighty times compared to accessing data from system memory. Cache efficiency is the performance linchpin to most modern single and multi-processor systems.

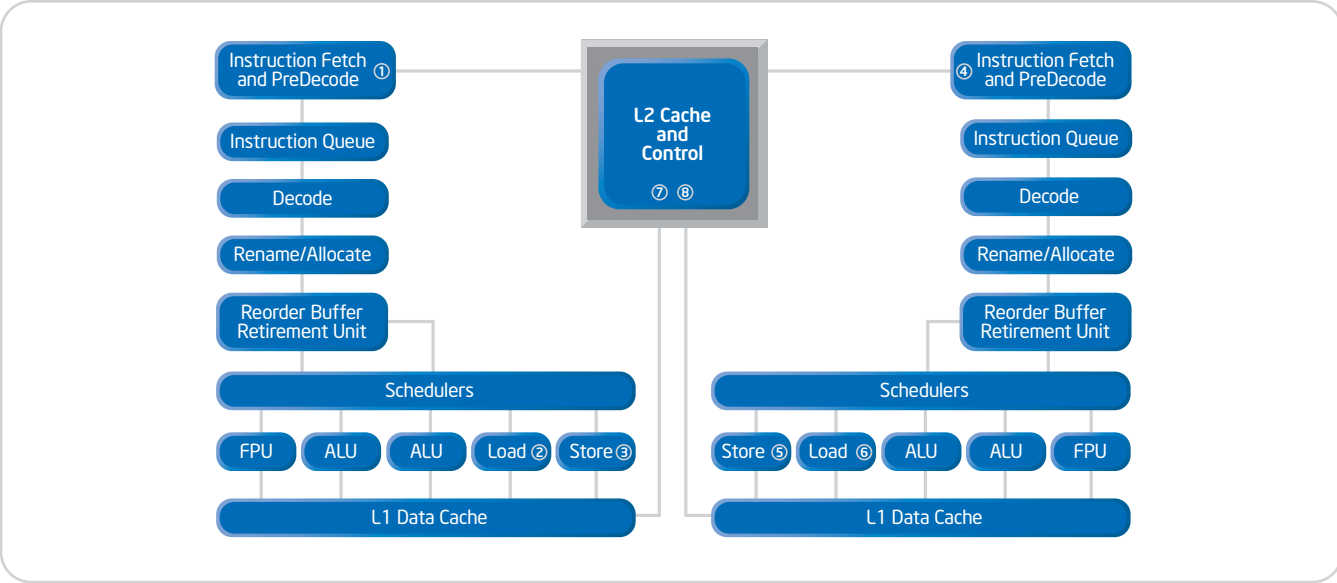

Figure 1. The Intel® Core™2 Duo processor

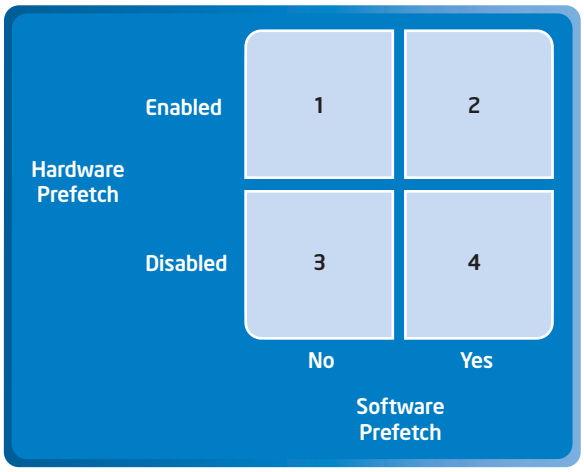

Figure 2. Four data prefetching scenarios

Cache Efficiency and Memory Access Stride

As mentioned previously, data prefetching is a technique used to increase cache efficiency and hide memory latency. A hardware prefetch unit observes the pattern of memory accesses issued by the execution pipeline and attempts to predict what accesses will come next. It then prefetches the data from this memory location so it will be in the cache when the execution pipeline requests it. A common technique for predicting future memory accesses is to observe the distance in address space between two memory accesses. This distance is commonly called the "stride." When the stride becomes long, the hardware prefetch unit has more difficulty predicting what memory locations to retrieve. Since hardware prefetching does not cross page boundaries, it is most effective for small-stride accesses.

For a given application, if hardware prefetching isn't particularly effective, it is possible to proactively load data into the caches using software prefetch instructions. This way, the application developer can instruct the CPU to retrieve data from system memory before it is needed in future computations.

It is interesting to note that a prefetching scheme may be optimal for one piece of code, yet sub-optimal for another piece of code in the same application. AMIDE is an open-source medical imaging application that illustrates this disparity. When AMIDE searches an array using short-stride memory accesses, the hardware prefetcher provides the best results. However, when AMIDE accesses memory with long-strides, software prefetching instructions deliver greater performance.

To quantify the impact of prefetching on the AMIDE application, this paper presents performance data for different combinations of hardware and software prefetching, as shown in Figure 2.

Medical Imaging and Data Prefetching

The data prefetching benchmarks were collected on AMIDE (A Medical Imaging Data Examiner) which is a free, open-source tool for viewing and analyzing medical images. The following URL contains more information on AMIDE: AMIDE.sourceforge.net. AMIDE also utilizes the VolPack Volume Rendering Library and this software is covered by the Stanford Computer Graphics Laboratory's General Software License.[1]

AMIDE supports a range of features including some basic commands for rotating images, selecting layers to view, and fusing multiple images. A screen shot from AMIDE is shown in Figure 3.

AMIDE searches a medical imaging dataset and renders 2-D images for display on a PC monitor. In this paper, AMIDE performance data is measured by the time required to render an image after the user requests to view the medical image from a new angle. Depending upon the angle of rotation, AMIDE decides whether to search the medical image dataset from the X-, Y- or Z-axes as shown in Figure 4.

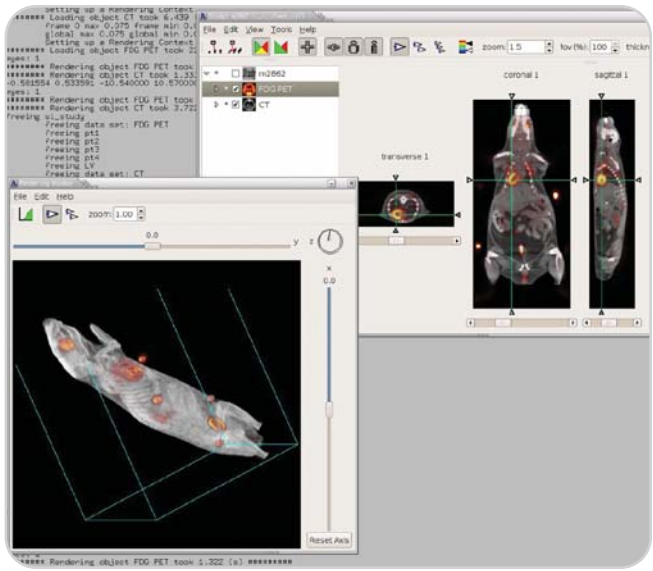

Figure 3. Image rendering by AMIDE*

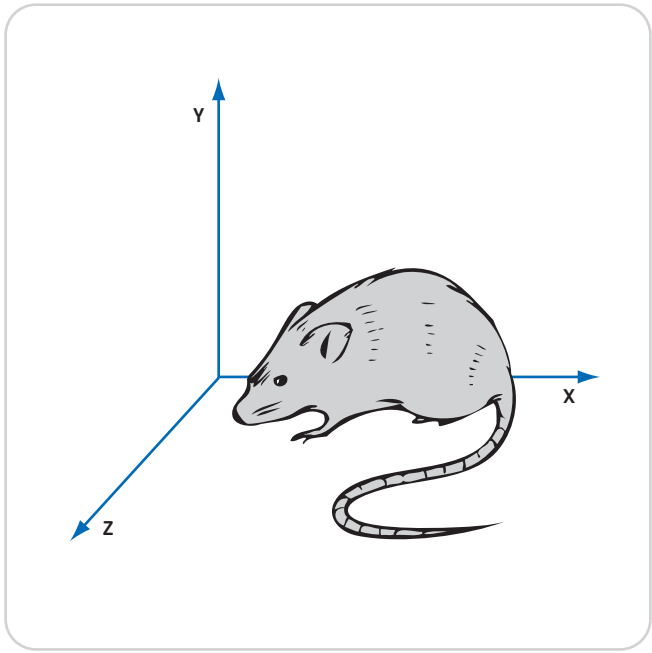

Figure 4. Searches are along X-, Y- or Z-axes

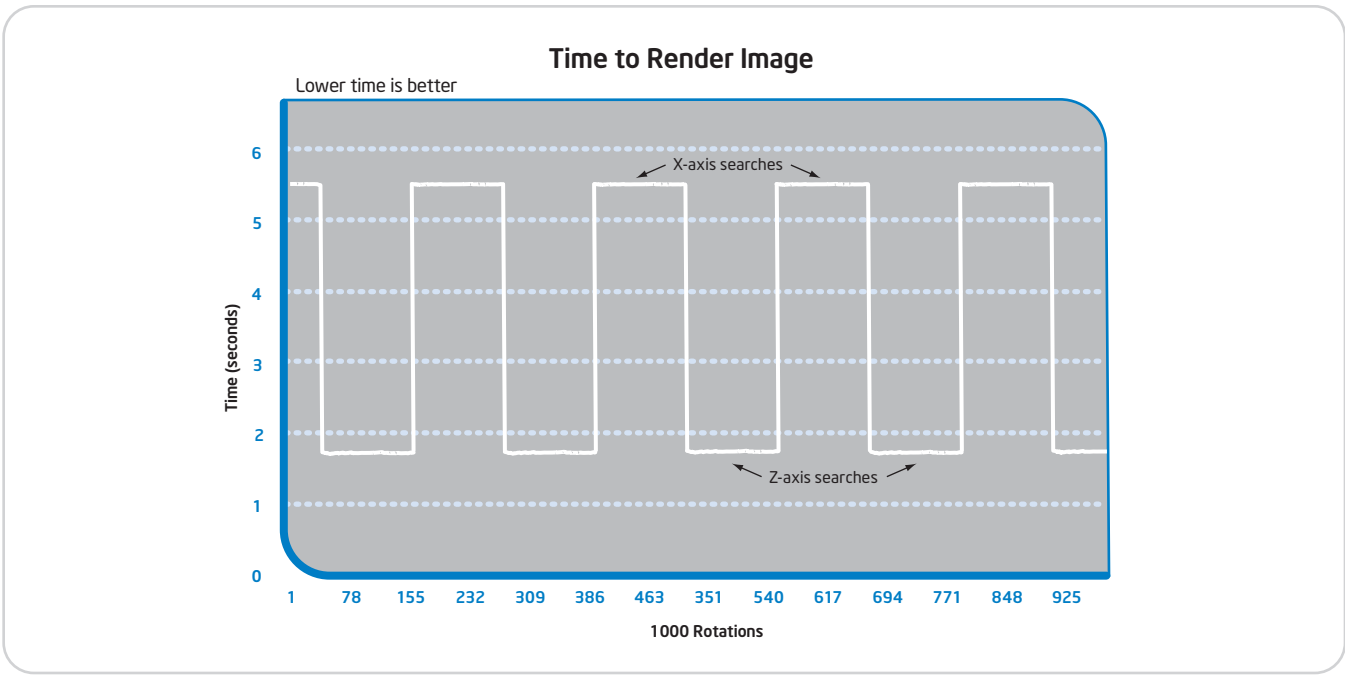

Figure 5. Time to render an image for 1000 rotations

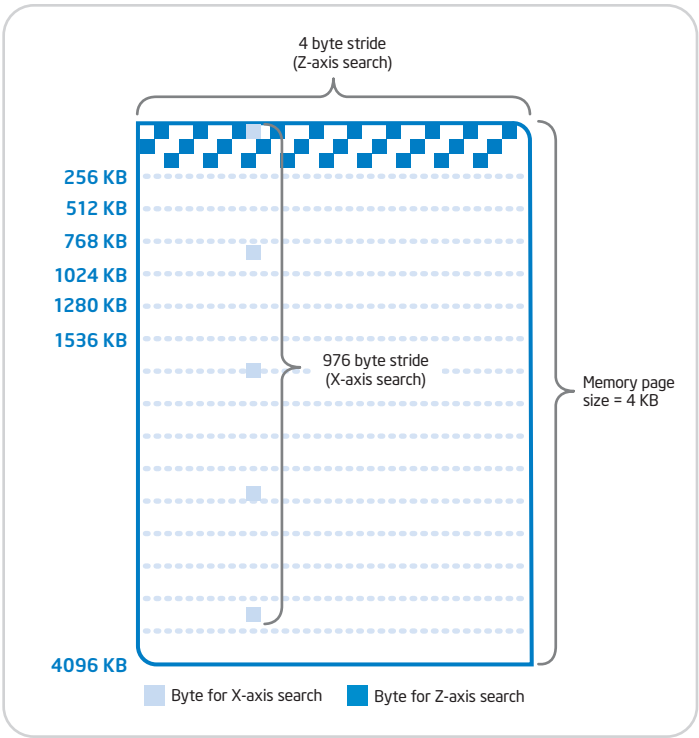

Figure 6. X- and Z-axes memory access strides

The time AMIDE requires to display an image is dependent upon the axis selected to traverse. For example, a search along the Z-axis tends to be roughly three times faster than along the X-axis, illustrated for 1000 rotations in Figure 5.

AMIDE loads a medical imaging dataset structured as a one-dimensional array. When AMIDE searches the dataset along the Z-axis, bytes are accessed sequentially with a stride of four bytes which is a relatively short stride, as depicted by the dark squares in Figure 6. For the X-axis, accesses have a stride of 976 bytes (light squares) which is relatively long and results in more memory paging overhead. In other words, with a stride of 976 bytes, only four or five memory accesses are possible before a page fault occurs. These strides, 4 and 976 bytes, are specific to the AMIDE algorithm and the way it transforms 3-dimensional data into a one dimensional array (data structure).

```
void prefetch function(int
base address) {
    int address = base address;

    asm("push %edx;");
    asm("movl %0,%%edx"
         :
         :"r"(address)
         );

    asm("prefetcht2 (%edx)");
    asm("pop %edx;");

    return;
 }
```

```
while (count > 0) {

prefetch function(topRLEdata +
4*voxel istride);
prefetch function(botRLEdata +
4*voxel istride);

.........}
```

Figure 7. Code example

Four Data Prefetching Scenarios

Four data prefetching scenarios can be generated by enabling/disabling hardware prefetching and including/excluding software prefetching instructions. Hardware prefetch is controlled by the BIOS and is normally enabled by default. Typically, hardware prefetching is a manual BIOS setting so developers can easily enable or disable this feature.

CPUs often support software prefetching instructions. These instructions load various combinations of L1, L2 and L3 caches. The code segment above employs an instruction called prefetcht2 which loads referenced data into the L2 cache using the prefetch_function code segment shown above. This function receives the address of the data that requires prefetching.

The code calling the prefetch_function is shown in figure 7. This code snippet, where topRLEdata and botRLEdata are pointers to the data that will be rendered. The stride of the memory access from one loop iteration to the next is stored in voxel_istride. The software prefetch instruction references data needed in future iterations of a loop; the code example references data four loop iterations in advance. This gives the data enough time to arrive in the L2 cache before it actually is used. For this code, prefetching data four loop iterations ahead produces better results than 2, 8 or 16 loop iterations ahead. The placement of the prefetch instructions was chosen empirically, through experimentation, in order to take into account the many system factors that come into play.

Performance Results

The impact of using hardware and software prefetching varies within a medical imaging dataset. Renders along the Z-axis are three times faster than along the X-axis using *"Hardware Prefetch/ No Software Prefetch."* The other three prefetching scenarios are all slower for renders along the Z-axis as shown in the first row of data in Figure 8. The worst case is *"No Hardware Prefetch/Software Prefetch"* which is 25% slower.

However, the image renderings along the X-axis perform best with *"No Hardware Prefetch/Software Prefetch"* which decreases the render timer by 70% compared to *"Hardware Prefetch/ No Software Prefetch."*

From an application perspective, improving the worst-case renders by 70% at the expense of degrading the best-case renders by 25% is a good trade-off. This may be more evident after reviewing the render time data in Figure 9 on the next page. The best average performance occurs for the *"No Hardware Prefetch/Software Prefetch"* scenario.

The comparative effectiveness of the various prefetch scenarios can be seen clearly in Figure 10 on the next page, which plots the average render time for the four data prefetch scenarios for the X- and Z-axes searches. The performance rank of the data prefetch scenarios for the X-axis searches is the inverse of the rank for the Z-axis searches. This performance comparison shows that a prefetching scheme which is most effective for one piece of code may be sub-par for another code block in the same application.

It is possible to add software prefetching instructions in code segments where it increases performance. Software prefetching can also be turned on/off. Using the AMIDE study as an example, if the program detects the stride of the memory is short (e.g., 4), then it can decide to jump over the software perfecting instructions. Conversely, if the program detects a long stride (e.g., 976), then it can execute the software prefetching instructions. The physical address pointer can be used to determine the stride of the memory accesses.

Unfortunately, it is not practical to control the hardware prefetcher dynamically depending upon which code segment is running at the time. Since hardware perfecting is typically determined at system boot, it is not possible to enable/disable hardware prefetching while the application is running.

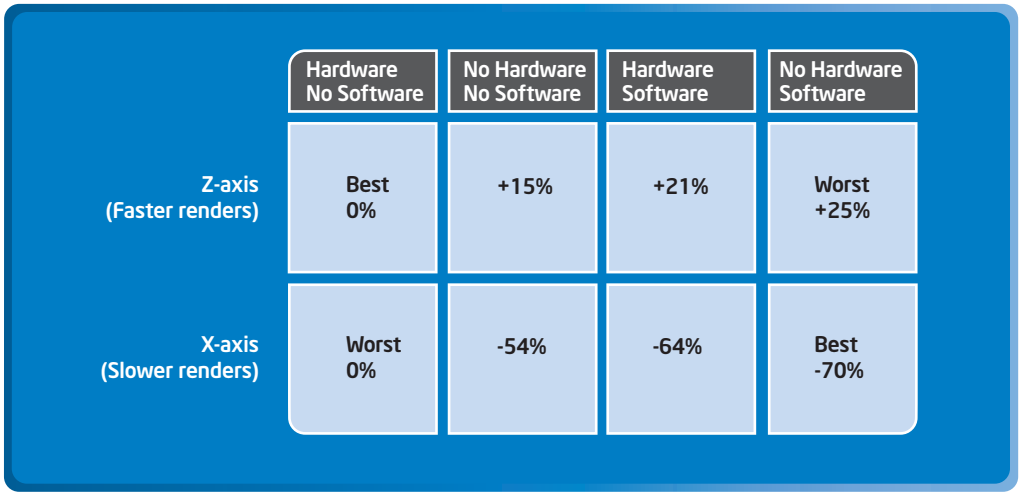

Figure 8. Comparison of rendering times for four data prefetching scenarios (times are relative to *"Hardware Prefetch/No Software Prefetch"* scenario)

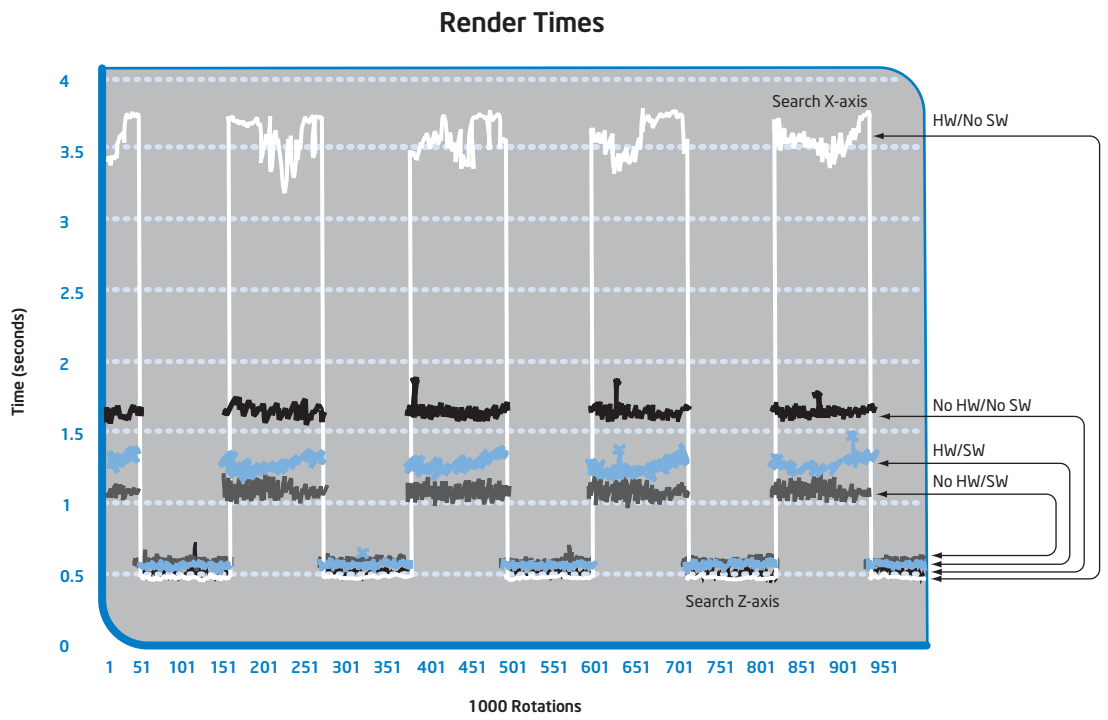

Figure 9. Time to render for four data prefetch scenarios for 1000 rotations

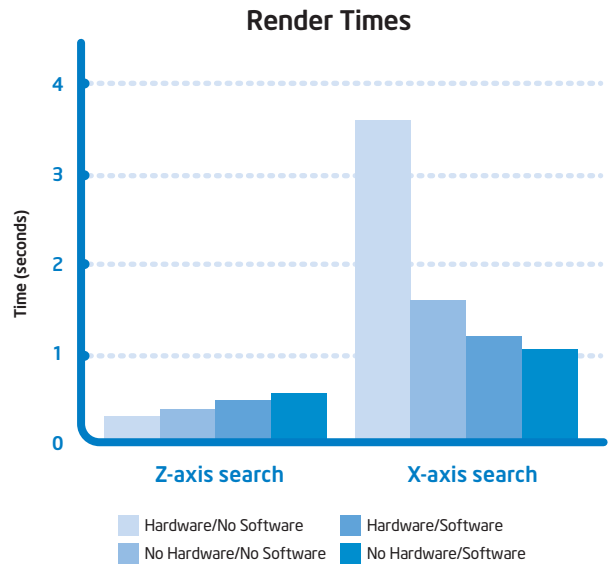

Figure 10. Average time to render for four data prefetch scenarios

Conclusion

This paper showed how the performance of a medical imaging application can vary dramatically (three-fold) for different data prefetch implementations. The data suggests that average performance of AMIDE is improved when hardware prefetching is disabled and software prefetching instructions are added.

The results of the AMIDE study can be applied to many other embedded applications which are finely tuned for particular workloads. To tune performance, embedded system developers may find new avenues to increase performance by controlling how their systems prefetch data.

[1] We would like to extend our appreciation to Seven Pinnacles developers Diana Franklin and John Seng, who also serve as professors at The California Polytechnic State University, San Luis Obispo, for their contributions to this benchmarking activity.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations.

Printed in USA 0906/AV/OCG/XX/PDF ♻ Please Recycle 315256-001US